Implementation Issues in Product Line Scoping

Klaus Schmid and Cristina Gacek
Fraunhofer Institute for Experimental Software Engineering (IESE)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{schmid, gacek}@iese.fhg.de

ABSTRACT
Often product line engineering is treated similar to the waterfall model in traditional software engineering, i.e., the different phases (scoping, analysis, architecting, implementation) are treated as if they could be clearly separated and would follow each other in an ordered fashion. However, in practice strong interactions between the individual phases become apparent. In particular, how implementation is done has a strong impact on economic aspects of the project and thus how to adequately plan it. Hence, assessing these relationships adequately in the beginning has a strong impact on performing a product line project right.

In this paper we present a framework that helps in exactly this task. It captures on an abstract level the relationships between scoping information and implementation aspects and thus allows to quickly analyze implementation aspects of the project. We will also discuss the application of our framework to a specific industrial project.

Keywords
Software product line, domain engineering, scoping, architecting, implementation

1 Introduction

Recently, the importance of an architecture-based approach to software reuse has been recognized as being a key driver to achieving high reuse levels. Product Line Software Engineering combines this with ideas from domain engineering [1, 2] like the up-front analysis of commonalities and variabilities present in the product line. A specific method for Product Line Software Engineering is PuLSE™, an approach developed at the Fraunhofer IESE [3]. This approach provides technologies for addressing all life cycle stages of a product line engineering project: scoping, domain analysis, architecture development, and system implementation. In this paper we will concentrate on the approach proposed for scoping reuse infrastructures in the context of product line engineering. This approach is called PuLSE-Eco [4].

PuLSE-Eco is based on the idea of using the business objectives in deriving the product line scope. That is, they are the key criteria for deciding whether something should be part of the scope or not. Thus, depending on the specific business objectives relevant to the company, the scope may vary, even for the same line of products.

As part of our exposure to industrial projects we had to recognize that the different phases of product line development are strongly interrelated with each other. Already during scoping some implementation level information is needed in order to adequately analyze the economics of the product line project that has to be planned. This fact lead us to analyze more deeply the relationship between the general product line situation and the implementation aspects. In this paper we present the results of this analysis.

The paper is structured as follows: in the remainder of this section we will discuss in more detail the type of relations between scoping and implementation we are looking at in this paper. In Section 2 we will describe in some detail our framework. First, we will discuss in Section 2.1 the environmental factors on which we base our framework and in Section 2.2 the implementation aspects for which we want to derive some information. Section 2.3 will then describe in some detail the relationships we found. In Section 3 we will then apply this framework to a case study and describe the insights we could gain from this. Finally, Section 4 concludes.

1.1 Product Line Scoping

Scoping is a key step in any product line or domain engineering project. It determines actually which parts of the systems will be supported by reusable assets:

- If too much is supported in a reusable way, this may impair the overall pay-off as engineering for reuse is usually more expensive than single-system development.
- If too little is supported, the situation may actually be even more complicated, as assets that do not support the necessary range of variability can only be used in a subset of the required products.

This shows that it is both very important to do scoping right and at same time not easy to do it right.

In particular, scoping impacts the following sub-decisions:

- Should we build a product line for this particular system family?
- What functionalities to include in the domain analysis?
- What functionalities should be directly supported by the reference architecture?
  (e.g., by a component of the architecture)
- For which functions should reusable lower level assets be implemented?

Often scoping is performed to analyze whether a certain domain should be supported in terms of a reuse infrastructure. This view is especially common with domain engineering approaches [1, 2, 5]. This is typically extended in a straightforward manner to product lines, i.e., the union of the functionality of the systems is regarded as “the domain”. PuLSE-Eco underlies a somewhat different concept. Domains are concep-

tual domains that describe a certain type of functionality (e.g., report writing, internet data transfer, etc.) several of them may be relevant to a single system. This situation is depicted in Figure 1.

1.2 Scoping Impacts Architecture and Implementation

As described above some important decisions for the reference architecture are already determined during scoping, strongly influencing the range of functionality supported by the reference architecture. While later activities may change this scope based on additional information and better insight gained during the latter stages, usually the basic scope will remain intact. Among the architecture decisions that are usually (implicitly) made during scoping are the following:

- The functionality explicitly supported by the reference architecture is roughly defined. This has the consequence that certain functionalities, although part of the product family, will not be explicitly supported by the reference architecture.
- The major variabilities that are explicitly supported by the reference architecture are identified. In particular this entails some minimum requirements on component interfaces (e.g., what functionality to encapsulate, whether the absence of components needs to be handled as is the case for optional features)
- Similarly, already some basic services are identified that will be relevant to all the systems and will thus be of general importance.

Further, some constraints are identified, which are not so obvious:

- Initial assumptions about the underlying architecture are made since they determine costs and effort estimates
- An initial guess about the major architectural style can be made as the major domain characteristics are known (e.g., event-based architecture, layered architecture)
- Assumptions on the implementation schema need to be made (e.g., generator, component-based, etc.) as the economies of the product line and thus the decision on what to support by reuse strongly depend on this (e.g., for generator-based implementation the up-front investment is much higher, but the costs of individual system is lower than the nominal case).

These decisions can be made that early as they largely depend on easy to elicit aspects of the product line like: number of systems, maturity of the domain, etc. However, while these assumptions are assumed to be incorporated in the product line reference architecture, in situations where one or more of these assumptions cannot be properly or efficiently accommodated in the reference architecture, the results of scoping must be revisited accordingly.

1.3 Architecture and Implementation Impact Scoping

While the fact that scoping has an impact on the definition of the reference architecture seems to be rather straightforward, the fact that implementation level assumptions (e.g., architecture) are relevant to identifying the appropriate scope is less obvious. However, one such influence has just been illustrated: in order to derive the scope and the economic viability of a scope, assumptions about the implementation technology need to be made. However, some more obvious uses of an architecture are:

- An existing architecture (e.g. for a single system which shall be expanded towards a product line) needs to be taken into account as it will influence the future architecture (cf. [6]).
- Even in cases where no system architecture exists there will be inherent assumptions in defining the features of the system and what is exactly meant by a certain functionality. These include things like whether the feature definition encompasses all the activities down to the data base access or whether intermittent features exist that perform some pre-processing. Initial decisions and structuring of interacting features of this type are called a *conceptual architecture*. As the conceptual architecture obviously influences what scope definitions are made, it is better to make it explicit.
- Especially for identifying the economics of the scope, it is important to make the conceptual architecture explicit, because an estimate on the amount of effort needed for a feature can only be given after it has been clarified what behaviour shall be regarded as part of this feature.
- The way the product line is supposed to be implemented is very important to determining the economics of the scope definition, hence, what is the most appropriate scope.

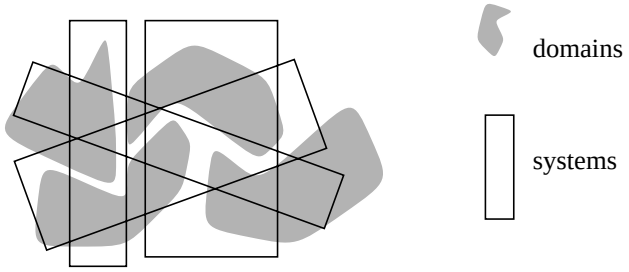

Figure 1. Relationship between domains and systems

As described above, there are quite a few aspects that become visible during scoping, which lead to serious constraints on the architecture and the implementation of a software product line. In turn, these constraints and the solutions chosen have actually a large impact on the economics of the product line situation and thus on how the scope should be derived. Consequently, they need to be made explicit as early as possible, just like the conceptual architecture.

In order to do so, we developed a framework that makes the relationship between the high level aspects (e.g., business goals), the architectural issues, and chosen implementation method explicit. As product line development is still in its early days and only little validated information on this relationship is documented, we could not rely as much on experience as we would have liked to. However, quite a few relations can be deduced from technological knowledge in the field with sufficient confidence. Additionally, we have been able to partially validate our framework in the context of an industrial project. At this point we have to focus on this type of information for the framework presented here. The following benefits are expected of the framework:

- Help in performing the scoping activity right, as it allows to deduce relevant information.
- Restrict (in an adequate way) the solution space for the implementation aspects of the product line, similar to the framework provided in [7] for architectures based on domain characteristics.
- Support validation of existing reference architectures.
- Be a starting point for aggregating further knowledge, as more and more lessons learned are found.

The relationships are briefly summarized in Figure 2.

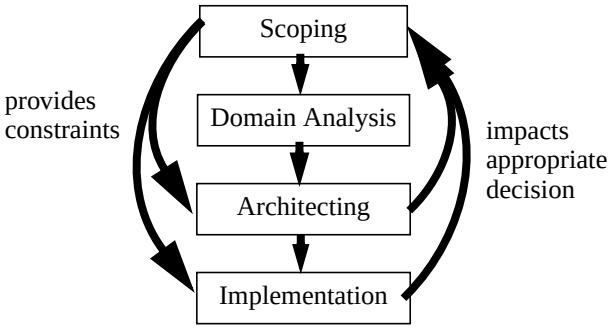

Figure 2. Relationship between scoping and implementation

1.4 Related Work

The idea that high-level inputs have important implications for architecture and implementation decisions is not new, to the contrary [7]. Consequently, we will not reiterate this type of material. Instead we will concentrate on PL aspects.

Within the realm of scoping approaches we do not know at this point of any other approach which tries to explicitly fill the gap of determining which implementations assumptions should be used in the scoping process. Some assessment-based approaches to scoping like [5, 8] use criteria like maturity of the domain directly to assist in scoping, however, they make the feedback loop which exists not explicit. On the other hand economic models [9] usually use the implementation characteristics we identified here in order to determine the reuse project economics.

A framework for assisting an architectural style choice based on certain domain characteristics has been previously defined [10] and later refined [7]. That work provides a characterization of architectural styles based on a set of features focusing on control and data issues. It also provides rules-of-thumb for architectural styles selection while considering the required characteristics of the architectural solution being built. vs. the architectural styles intrinsic characteristics. Although their work is quite useful, it addresses the construction of one-of-a-kind software system architecture only. These considerations are also relevant for the construction of product line reference software architectures, but are not enough. There are many product line environmental factors that must also be considered.

Classifications of reuse approaches have existed for a while [11, 12]. These usually compare and contrast various approaches, yet fail to provide guidelines on their selection depending on the situation.

In this paper we provide support for architectural style selection within product lines, as well as define clear guidelines for the resolution of reuse infrastructure implementation decisions. The underlying approach that is used for selecting a specific technology for a specific situation is similar to the selection of technology packages in the context of experience factories [13].

It is important to note that architectural style choice is just one of the issues relevant to the definition of a product line reference architecture [14]. Other considerations are less dependant on the product line environmental factors, hence not in the focus of this paper. These other considerations are addressed elsewhere [15, 16].

2 The Framework

In this section, we will describe the basic framework for relating business factors and implementation factors. Note, however, that we did refrain from including all possible such relationships. Instead, we concentrated on those aspects that are particular to product lines, as other aspects like the relationship between domain aspects and architecture have been described elsewhere [7]. Below, we will first discuss the business (or environmental) factors and the reasons for choosing these particular factors. Then, we will similarly discuss the implementation level aspects. In a third subsection we will then describe the relationships we found.

2.1 Environmental Factors

There are quite a few characteristics of a product line project that can be easily surveyed with the help of domain experts early on with little effort and have a quite strong impact on the technical solutions one should be aiming for. Here we will

describe the ones we could identify so far both from literature [5,1], as well as by our experience and thorough study of the topic area. Below we will discuss each of the factors we could so far identify as being relevant. Note that the scales we propose are clearly subjective. This is not assumed to be a problem since only the magnitude of the values is relevant.

Number of independent features
How many features relevant to distinguishing the various members of the product line can be identified? The measure is relative to the overall size of the functional area. Meaning larger functional areas can also be expected to have more features without changing the value of the measure. The scale has the values low, medium, high (e.g., for a domain estimated at 100 kLoC 10 features would be low, while 100 would be high).

Structure of the product line
This captures whether variabilities among systems are dominated by optionality or alternative. Usually, both of them will exist, thus we are looking here at the predominant type of variability. Scale is: optional, neutral, alternative (e.g., 20% optionalities, 80% alternatives would still be captured as alternative).

Variation degree
What percentage of a system is expected to be covered by the core (i.e., the overall common) part? low, medium, high (low ~ 40%, high ~80%).

Number of products
What is the number of products the product line is expected to contain? Scale: low, medium, high (low<=5, high>=12)

Complexity of feature interactions
This describes how interrelated features are on average. Two features are called interrelated if one modifies the behaviour of the other (i.e., the functionality is not just the sum of the two). This is again measured by low, medium, high.

Feature size
The size of a feature is basically the amount of code relevant to implementing it. It is measured on a scale ranging from low (approx. one procedure/method/object) to high (a complete subsystem).

Performance requirements
The performance requirements (memory, run-time) are measured relative to what is not easy to provide. Thus, the performance requirements are called strict, if they are expected to be a high priority design rationale to squeeze out the required performance level. Otherwise (i.e., it is obvious that the required performance levels can be achieved) the performance requirements are called loose.

Coverage
This basically measures to what extent the potential feature combinations will actually occur. For example, if 100 optional features exist then the domain contains 2^{100} possible combinations; if actually only a small number of products (10) will be developed than the coverage is obviously low. Conversely for high coverage.

Maturity/Stability
If the domains relevant to the product line are not expected to change and are well understood (e.g., as shown by standardization) then they can be regarded as being of high maturity/ stability. Scale: low, medium, high

Entry points
Three different starting situations can be distinguished for the product line project (cf. [17]):

- Independent PL: a new product line is developed from scratch
- Integrating PL: product line is introduced while some products are already under development
- Reengineering-driven PL: the core product line assets are reengineered from legacy systems

Openendedness
This describes the range of functionality that may be relevant to the systems now and in the future (i.e., can it be expected that the currently identified set of features will also cover future systems well or is there an expectation that future product line members may need other features?). As opposed to maturity/stability this doesn't address the change in the features that are relevant to a domain, but with respect to the domains that are relevant to the system family. Scale: open, neutral, bounded.

2.2 Implementation Aspects

Similarly to our discussion of the environmental factors as input to the framework, we are now going to describe the aspects we want to derive values for as results from our framework. So far we could identify four aspects for which recommendations (i.e., constraints) can be derived from the input factors described above.

Each of the four aspects can be seen as independent in the sense that the values for the various factors can be independently derived and used. Further, especially the values for the first three categories should be seen as describing a continuum, within which only some extreme have been identified similar to the environmental factors.

2.2.1 Type of reuse infrastructure

What kind of technology should be the basis for the resulting infrastructure construction (i.e., what is the aimed-at result)? At this point we distinguish three different categories.

Software Platform
While in the literature many different meanings are given to the term *software platform*, we use it here explicitly to refer to groups of assets that only address the commonalities within a product line, i.e., the assets contained in the software platform will be incorporated in every product in the product line.

Product Line/Reference Architecture
As opposed to a platform a reference architecture provides the concepts for the complete products, including also variabilities.

Domain-Specific Language (DSL)
A DSL is provided that completely abstracts from all implementation details and covers all characteristics that may be relevant to systems.

2.2.2 Variability Representation

This describes how variability is mapped to code from an implementation point of view.

Pure Code
The code assets are expected to contain only code. No parametrization except for the selection of code components and the run-time parameters are expected to exist.

Parametrization
In this case some compile-time parameters are expected to exist. Conditional compilation is one approach for implementing this approach.

Template
In this case, assets are more generalized and code fragments may be reassembled in a rather sophisticated way during compilation. Frames or aspect-oriented programming are examples of this kind of approach.

DSL
Here, the full capabilities of a language can be used to describe variabilities in the domain. (Note that opposed to the previous section here we use DSLs not as a means to describe the coverage, but as means to describe variability.)

2.2.3 Level of Detailedness

As [14] points out, there are many different interpretations of what level of detail is implied by the term product line architecture. Similarly, many different interpretations can be given to the term reuse infrastructure. Here, we simplify the discussion by distinguishing only three main levels of detailedness of the reuse infrastructure. (Below code is meant to include also templates and parametrized code.)

Reference Architecture Description
Only a reference architecture description is developed but no code is actually produced.

Core Code
On top of the architecture description code components to cover the core functionality are developed.

Full Range
Here, most code components (except for system-specific parts) are actually developed.

2.2.4 Architectural Concept

Unlike the other implementation aspects, the architectural concept cannot be precisely defined based only on the environmental factors described earlier. The style choice for the overall product line reference architecture and its subsystems depends also on external factors. The most influential external factors that must be taken into account are domain drivers, pre-existing architectures of systems and/or subsystems legacy or already under development, as well as the architectural assumptions made while scoping was being performed.

How domain drivers impact the architectural style choice has already been described by Mary Shaw and Paul Clements [10]. During the construction of a product line reference architecture, just as for single system architectures, their approach for selecting an architectural style should always be taken into account.

Additionally, if the entry point is reengineering-driven or integrating product line, pre-existing architectures or architectural parts do heavily influence the architectural style choice. The reason for this is twofold, the existing architecture already has its own style [18], and, while using existing parts, architectural mismatches must be either avoided or handled appropriately [19].

The impacts suggested by the environmental factors discussed in section 2.1 will be discussed shortly.

2.3 The Relationships

Above we described both environmental factors and implementation aspects for software product lines. However, as we discussed previously, there is a close relationship between the identified factors. The relationships we could identify so far are summarized in this section. They had to be derived mostly from our background on the subject matter as to our knowledge so far no comparable studies exist. Consequently, these relationships should be regarded as being of preliminary nature.

Each of the relationships will be described below in the form of a prototypical situation (in terms of the environmental factors) in which the corresponding value for the implementation factors should be chosen. As a real situation will usually not correspond exactly to a certain prototypical situation, the results may not exactly correspond to one of the prototypical results. A further discussion on how to use the results of applying the relationships in a real situation is given in Section 3.

2.3.1 Type of reuse infrastructure

Software Platform
A software platform, especially in the sense of a platform that supports several product lines (e.g. cellular phone, as well as wired phones), is usually a high investment that is only worthwhile, if a large number of products will be supported by it. In order for such an investment to pay off also a high stability and maturity of the domain is required.

1)	Number of independent features:	*
2)	Structure of the product line:	*
3)	Variation degree:	low..medium
4)	Number of products:	high
5)	Complexity of feature interactions:	low
6)	Feature size:	*
7)	Performance Requirements:	loose

8) Coverage: low
9) Maturity/Stability: high
10) Entry Points: *
11) Openendedness: *

Reference Architecture
Here, the idea is to get a single line of products under control. This is worthwhile if overall variations are restricted, so that a common architecture can be realistically provided (i.e., all systems have a similar structure).

1) Number of independent features: *
2) Structure of the product line: *
3) Variation degree: *
4) Number of products: *
5) Complexity of feature interactions: low..medium
6) Feature size: medium..high
7) Performance Requirements: *
8) Coverage: low..medium
9) Maturity/Stability: *
10) Entry Points: *
11) Openendedness: *

DSL
A complete coverage of all software that might be possibly relevant to the product line is only worthwhile if the problem domain is clearly bounded and stable and a large number of products is expected to exist such that a positive return on investment can be expected.

1) Number of independent features: low..medium
2) Structure of the product line: *
3) Variation degree: medium..high
4) Number of products: high
5) Complexity of feature interactions: medium..high
6) Feature size: low..medium
7) Performance Requirements: loose
8) Coverage: medium..high
9) Maturity/Stability: medium..high
10) Entry Points: independent
11) Openendedness: bounded

2.3.2 Variability Representation

Pure Code
This is meaningful in particular if features do not have a large impact on the implementation of other features and if only a small subset of these combinations will actually be implemented.

1) Number of independent features: *
2) Structure of the product line: opt
3) Variation degree: *
4) Number of products: *
5) Complexity of feature interactions: low
6) Feature size: *
7) Performance Requirements: *
8) Coverage: low
9) Maturity/Stability: *
10) Entry Points: *
11) Openendedness: *

Parametrization
This allows to represent more variabilities and interactions, but has the down-side to require a higher effort and is more difficult to do right, which implies that the domain(s) should be rather stable and mature.

1) Number of independent features: *
2) Structure of the product line: alt
3) Variation degree: *
4) Number of products: *
5) Complexity of feature interactions: low..medium
6) Feature size: *
7) Performance Requirements: *
8) Coverage: low..medium
9) Maturity/Stability: medium..high
10) Entry Points: *
11) Openendedness: *

Templates
This again allows for a larger degree of variation but is again more difficult to do and requires a larger number of systems to pay off.

1) Number of independent features: *
2) Structure of the product line: *
3) Variation degree: medium..high
4) Number of products: medium..high
5) Complexity of feature interactions: medium..high
6) Feature size: low..medium
7) Performance Requirements: loose
8) Coverage: medium..high
9) Maturity/Stability: medium..high
10) Entry Points: independent, integrating
11) Openendedness: bounded

DSL
Again a larger degree of variabilities can be represented, but again also a larger effort is required. As the underlying economics are the same as in Section 2.3.1, the same situational characteristics apply.

2.3.3 Level of Detaildness
How much up-front investment is meaningful mainly depends on how strongly the systems will overlap and thus how much of the investment can be recovered over the various systems. Scoping itself will then be used to refine this a-priori expectation.

Reference Architecture Description
One will restrict oneself usually to this solution if the situation is rather unclear and only few systems are expected to recover the up-front investment.

1) Number of independent features: *
2) Structure of the product line: *
3) Variation degree: low
4) Number of products: low
5) Complexity of feature interactions: *
6) Feature size: *
7) Performance Requirements: *
8) Coverage: low
9) Maturity/Stability: medium
10) Entry Points: *
11) Openendedness: *

Core Code
In case variability is high, thus covering all combinations will not pay off, but the core can be expected to be sufficiently stable this approach can be assumed to be adequate.

1) Number of independent features: *
2) Structure of the product line: *
3) Variation degree: low..medium
4) Number of products: low..medium
5) Complexity of feature interactions: *
6) Feature size: *
7) Performance Requirements: *
8) Coverage: low
9) Maturity/Stability: medium
10) Entry Points: *
11) Openendedness: *

Full Range
If variability over the whole range of product features is sufficiently stable and under control, and covered by products, this approach can be regarded as most adequate.

1) Number of independent features: *
2) Structure of the product line: *
3) Variation degree: medium..high
4) Number of products: medium..high
5) Complexity of feature interactions: low..medium
6) Feature size: *
7) Performance Requirements: *
8) Coverage: medium
9) Maturity/Stability: medium..high
10) Entry Points: *
11) Openendedness: *

Table 1 gives a summary of the relationships we described above.

2.3.4 Architectural Concept

Given the fact that the environmental factors cannot clearly define directions to be considered by the reference architecture while ignoring external factors (see section 2.2.4), in this section we will simply describe how each of the environmental factors contributes to architectural decisions. The weight given to specific factors, both environmental and external, varies from situation to situation, depending on the major risks and priorities at hand. Consequently, we cannot prescribe here how the combination of differing influencing factors should be dealt with. A method such as ATAM [20] should be used for resolution support.

Number of Independent Features
When the number of independent features is high, the use of layering would be recommended simply because it provides higher levels of abstraction to facilitate reasoning.

Structure of the Product Line
Having alternatives or not has no impact in the architectural style choice. The impact is only on making sure that the various alternatives do implement the same interface and that their underlying assumptions do not clash with the rest of the architecture. The use of layering or abstract data types may help by localizing considerations on various alternatives.

The existence of optional items does have a stronger architectural impact. Components and connectors interacting with the optional parts must be able to handle both their presence and their absence. The best way to deal with optional architectural items is to use styles where components are self contained and ignore the existence of others, by assuming that required services will be performed somehow elsewhere. Examples of these styles are event based, blackboard, C-2, database centric, pipe and filters, and communicating processes.

Variation Degree
If the variation degree is low, the architecture for every system instance will have to have many components and connectors added as being system specific. Hence, one must already plan for ease of component and connection addition. The styles best suited for this purpose are event based, blackboard, C-2, database centric, pipe and filters, and communicating processes.

Number of Products
Integrating instance specific parts has inherent costs. Differing architectural styles may provide lower instance specific integration costs, yet higher set up costs (e.g., blackboard). The larger the number of products the product line is expected to contain, the better the justification for adopting such architectural styles.

Complexity of Feature Interactions
Unless this is extremely low, the styles mentioned in section should be avoided.

Feature Size
This factor has no architectural impact.

Performance Requirements
These are domain drivers that were considered by Shaw and Clements [10]. For considerations on this aspect please refer to their work [21].

Coverage
This factor plays only a very subjective role in the architectural context. If coverage is low, one must be careful not to over engineer the architectural solution.

Number of definable features	Structure of the product line	Degree of Variation	Number of Products	Complexity of feature interactions	Feature size	Performance requirements	Coverage	Maturity/Stability	Entry points	Openendedness	Type of reuse infrastructure	Representation of Variability	Level of Detailedness
*	*	*	high	low	*	loose	low	*	*	*	Plat-form	*	*
*	*	*	*	low/ med	med/ high	*	low/ med	*	*	*	Ref. Arch.	*	*
low/ med	*	med/ high	high	med/ high	low/ med	loose	med/ high	med/ high	indep.	boun ded	DSL	*	*
*	opt	*	*	low	*	*	low	*	*	*	*	Code	*
*	alt	*	*	low/ med	*	*	low/ med	med/ high	*	*	*	Param.	*
*	*	med/ high	med/ high	med/ high	low/ med	loose	med/ high	med/ high	indep/ integr.	boun ded	*	Templ.	*
low/ med	*	med/ high	high	med/ high	low/ med	loose	med/ high	med/ high	indep.	boun ded	*	DSL	*
*	*	low	low	*	*	*	low	low/ med.	*	*	*	*	Arch. Repr.
*	*	low/ med	low/ med	*	*	*	low	med	*	*	*	*	Core Code
*	*	med/ high	med/ high	low/ med	*	*	med	med/ high	*	*	*	*	Full Range

Table 1: Summary of the Framework Relationships

Maturity/Stability
A highly mature domain implies that there is a known (set of) solution(s) to the problem. A known architectural solution should then be used for the reference architecture (i.e., any style that can support the solution works).

A mature domain composed of subdomains that are still evolving can be best supported by layering or abstract data types.

A not so mature domain implies that components will be evolving, some extra ones be added, and existing ones removed. Best solutions to this are the styles event based, blackboard, C-2, database centric, pipe and filters, and communicating processes.

Entry Points
Both integrating and reengineering driven product line must consider pre-existing assets. The best way of dealing with pre-existing parts is by using layers, simply using the pre-existing architecture, implementing some wrapping scheme [18], and/or using instrumented connectors [21].

Openendedness
The values open and neutral imply that components will be evolving, some extra ones be added, and existing ones removed. Best solutions to this are the styles event based, blackboard, C-2, database centric, pipe and filters, and communicating processes.

3 Applying the Framework

In the previous section, we concentrated on a description of the relationships between the environmental factors and the implementation aspects we found. As we discussed earlier, these relationships can be read both ways. They are identified in the scoping phase, but have an impact on the implementation. On the other hand these implementation aspects need to be taken into account already during scoping. Here, we will concentrate on the first aspect and discuss how a description of a project environment can be used to derive a

first estimate of the implementation aspects.

The situation we discuss is derived from a real industrial project. A brief assessment of the situation leads to the following characterization of the product line situation:

1 **Number of independent features:** medium
2) **Structure of the product line:** opt
3) **Variation degree:** high
4) **Number of products:** low..medium
5) **Complexity of feature interactions:** low..medium
6) **Feature size:** low..medium
7) **Performance Requirements:** loose
8) **Coverage:** low
9) **Maturity/Stability:** medium
10) **Entry Points:** integration
11) **Openendedness:** open

As we discussed before, the domain description is not part of this characterization, as we concentrate here on those aspects that are specific to the product line approach, as opposed to those that are specific to the type of systems.

3.1 Implementation Aspects

When discussing the various relationships in Section 2.3, the aspects *type of reuse infrastructure*, *variability representation*, and *level of detailedness* were described in terms of prototypical situations. For determining the type of solution that is most appropriate to our example, we will use a simple similarity measure. The solution that has the highest similarity is then the one which is considered most appropriate for the situation. The fact that the different values we identified only specify specific points in a continuum, shows up in situations were two values have a similar rating. In these situations we assume that the "true" value lies somewhere in-between these values.

The similarity measure we will use here is very simply defined. We add over all attributes and divide the result by the number of non-'*' attributes. If a relationship says nothing about this attribute, then the value 0 is used. Likewise, if the value(-range) in the situation and the value (-range) in the relationship do not overlap. If the value(-range) in the situation is fully contained in the range determined in the relation a 1 is added. If there is only an overlap than 0.5 is added. This approach leads to the similarity values given in Table 2:

When we look at the results, we can make several interesting observations. For the type of reuse infrastructure the reference architecture has the highest value. This flows pretty well with our initial assumptions about the product line project, where based on the initial contacts we assumed (without the framework) that this would be appropriate.

For *representing variability* code has by far the highest value, with still a very high value for parametrization. Again this fits very well with our initial ideas about the product line. Additionally, the hint for parametrization — while originally we did not look at it — is regarded as an interesting idea that deserves more investigations, while the other options are clearly inappropriate.

With respect to the level of detailedness we can see that both *core code* and *full range* received similar values. This hints at the most appropriate resolution being somewhere in between these two values. Again, this flows pretty well with our intuition about the project that we strive towards a rather complete reusability of the code components, while some components may just not be appropriate.

After having identified this information we have a good starting point for performing a more reliable scoping, as we now have an informed estimate on how implementation will be done. This is crucial, as for example, the distinction between domain-specific languages, templates, and pure code has a strong impact on the overall economics of the product line project. Similarly the amount of code that will be shared has a strong impact. While it would be possible to derive values for each of the aspects from scratch whenever the situation arises it is very helpful to use this framework as it saves much time and it allows a neutral judgement (e.g., in the example above we would have made not always exactly these conclusions without the framework and wherever the framework provided a different hint, it was good to deeply consider the framework proposal.

3.2 Architectural Concept

Within this same project, we are currently working towards deriving a product line reference architecture. The fact that the structure of their product line is predominant on optionalities, is composed of domains with medium maturity, and is openended, suggest that styles such as event based, blackboard, C-2, database centric, pipe and filter, and communicating processes be considered.

Since the number of products expected to be in the product line is low to medium, a blackboard style is not really an option. Based on the domain drivers we have also been able to discard the usage of a pipe and filter style.

This product line is being built using a couple of pre-existing systems. We are currently in the process of retrieving their existing architectures. We already know that the overall systems are layered, but it is not yet clear which other styles

Table 2: Similarity Values for Case Study

Type of reuse infrastructure			Representation of Variability				Level of Detailedness		
Platform	Ref. Arch.	DSL	Code	Param.	Templ.	DSL	Arch. Repr.	Core Code	Full Range
0.625	0.8222	0.6	1	0.8	0.6666	0.45	0.625	0.75	0.7

are present where. Additionally, the decision on which parts to reengineer, which ones to redevelop, and which ones to incorporate as is has not yet been made.

All the factors above and the current product line requirements will be taken into account while we apply PuLSE-DSSA [15] and ATAM [20] to derive their reference architecture. This clearly identifies how environmental factors impact architectural decisions, yet also stresses that they are only part of the process.

4 Conclusions and Future Work

Due to little existing experience in product lines, guidance towards their key decisions is needed. In this paper, we highlighted the fact that information elicited during scoping also has clear impacts on architectural and implementation decisions. We have also shown that the converse relation, though less intuitive, also exists.

Our most important contribution here is towards defining a framework to support the resolution of implementation and architectural decisions based on environmental factors elicited while scoping is performed. We have illustrated how to use this framework with a real-life case study. This case study showed our approach to be very appropriate, providing a good fit and saving a substantial amount of time.

As future work we shall apply this framework to other environments, allowing us to further validate and improve the concepts as needed [22]. Additionally, we shall also provide a more formal mechanism for information capture and exchange between the scoping and architecting efforts, as well as specify configuration management rules and policies to be applied in this context.

5 Bibliography

1. *Reuse-Driven Software Processes Guidebook*, Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC, 1993
2. *Organization Domain Modeling (ODM) Guidebook*, Version 2.0, Software Technology for Adaptable, Reliable Systems (STARS), Technical Report STARS-VC-A025/001/00, 1996
3. J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. “PuLSE: A methodology to develop software product lines,” in *Proceedings of Symposium on Software Reusability’99* (SSR’99), May 1999.
4. J.-M. DeBaud and K. Schmid. “A systematic approach to derive the scope of software product lines,” in *Proceedings of the 21st International Conference on Software Engineering* (ICSE 99), 1999.
5. Department of Defense - Software Reuse Initiative, *Domain Scoping Framework*, Version 3.1, Volume2, Technical Description, 1995
6. J.M. DeBaud and J.F. Girard. “The Relation between the Product Line Development Entry Points and Reengineering,” in *Proc. of Workshop on Development and Evolution of Software Architecture for Product Families*, Las Palmas de Gran Canaria, Spain, Feb. 1998
7. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*, Addison Wesley, 1998
8. *Reuse Adoption Guidebook*, Software Productivity Consortium Services Corporation, 1993
9. J. Poulin. *Measuring Software Reuse*. Addison Wesley, 1997.
10. M. Shaw and P. Clements. “A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems,” in *Proceedings of COMPSAC 1997*, Washington, DC, August 1997
11. C. Krueger. “Software Reuse,” *ACM Computing Surveys*, vol. 24, no.2, June 1992, pp. 131-183
12. J.M. DeBaud. *The Construction of Software Systems using Domain-Specific Reuse Infrastructures*, Ph.D. Dissertation, Georgia Institute of Technology, Atlanta, GA, USA, 1996
13. Andreas Birk, Felix Kröschel. *A Knowledge Management Lifecycle for Experience Packages on Software Engineering Technologies*. IESE-Report No. 007.99/E, February, 1999
14. D. Perry. “Generic Architecture Descriptions for Product Lines,” in *Proceedings of Workshop on Development and Evolution of Software Architecture for Product Families*, Las Palmas de Gran Canaria, Spain, Feb. 1998, pp 51-56.
15. J. Bayer, O. Flege, and C. Gacek. “Creating Product Line Architectures,” submitted for publication
16. J. Bayer, C. Gacek, D. Muthig, and T. Widen. “PuLSE-I: Deriving Instances from a Product Line Infrastructure,” submitted for publication
17. J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid and T. Widen. *PuLSE™ — Product Line Software Engineering*. Fraunhofer Institute for Experimental Software Engineering. IESE-Report No. 020.99/E, 1999
18. J. Bayer, J.-F. Girard, M. Würthner, J.-M. DeBaud, M. Apel, “Transitioning Legacy Assets to a Product Line Architecture,” in *Proceedings of 7th European Software Engineering Conference* (ESEC) / *7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* (FSE), 1999
19. C. Gacek. *Detecting Architectural Mismatches During System Composition*, Ph.D. Dissertation, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781, USA, 1998
20. R. Kazman, M. Barbacci, M. Klein, S.J. Carriere, and S.G. Woods. “Experience with Performing Architecture Tradeoff Analysis,” in *Proceedings of the 21st International Conference on Software Engineering* (ICSE 99), 1999, pp. 54-63
21. R. Balzer. “An Architectural Infrastructure for Product Families,” in *Proceedings of Workshop on Development and Evolution of Software Architecture for Product Families*, Las Palmas de Gran Canaria, Spain, Feb. 1998, pp.158-160.
22. K.D. Althoff, A. Birk, S. Hartkopf, W. Müller, M. Nick, D. Surmann, and C. Tautz. “Managing Software Engineering Experience for Comprehensive Reuse,” in *Proceedings of the 11th Software Engineering and Knowledge Engineering Conference* (SEKE 11), 1999, pp. 10-19